Chalmers | GÖTEBORGS UNIVERSITET

Concurrent Programming TDA384/DIT391

Saturday, 26 October 2019

Exam supervisor: K. V. S. Prasad (prasad@chalmers.se, 0736 30 28 22)

(Exam set by K. V. S. Prasad, Sandro Stucki and Nir Pieterman,
based on the course given Sep-Oct 2019)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes; English dictionary.

Grading: You can score a maximum of 70 points. Exam grades are:

points in exam	Grade Chalmers	Grade GU
28–41	3	G
42–55	4	G
56–70	5	VG

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows:

points in exam + labs	Grade Chalmers	Grade GU
40–59	3	G
60–79	4	G
80–100	5	VG

The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

Instructions and rules:

- Please write your answers clearly and legibly: unnecessarily complicated solutions will lose points, and answers that cannot be read will not receive any points!

- Justify your answers, and clearly state any assumptions that your solutions may depend on for correctness.

- Answer each question on a new page. Glance through the whole paper first; six questions, numbered Q1 through Q6. Do not spend more time on any question or part than justified by the points it carries.

- Be precise. In your answers, try to use the programming notation and syntax used in the questions. You can also use pseudo-code, *provided* the meaning is precise and clear. If need be, explain your notation.

Q1 (8p). Below is the pseudo-code of a program with two threads, p and q. The variables n and flag are shared between p and q.

int n = 0; boolean flag = false;	
p	q
p_1: **while**(flag==false) {	q_1: **while**(flag==false) {
p_2: n = 1 - n;	q_2: **if**(n==0) {
}	q_3: flag = true
	}
	}

The labels p_1, p_2, q_1, q_2 and q_3 are given only for ease of reference.

(Part a) Construct a scenario for which the program terminates.*(2p)*

(Part b) What are the possible values of n when the program terminates? *(2p)*

(Part c) Does the program terminate for all scenarios? *(2p)*

(Part d) Does the program terminate for all fair scenarios? *(2p)*

Q2 (18p). The pseudo-code below tries to solve the critical section (CS) problem with two threads, p and q. The keyword **atomic** encloses a pair of assignments that happen in one step, i.e., no instruction by the other thread can occur between those two assignments. Remember that CS must exit after a finite time, but NCS may loop.

The label p_i can mean the command that follows p_i, or the proposition that thread p is at p_i, and *the next command* p *will execute is* p_i.

int tg= 1;			
p		q	
	int tp= 0;		**int tq= 0;**
	while(true) {		**while**(true) {
p_1	//NCS (non-critical section)	q_1	//NCS (non-critical section)
	do{		**do**{
p_2:	**atomic**{tp= tg; tg= 0};	q_2:	**atomic**{tq= tg; tg= 0};
p_3:	} **while**(tp!= 1);	q_3:	} **while**(tq!= 1);
p_4	//CS (critical section)	q_4	//CS (critical section)
p_5:	**atomic**{tp= 0; tg= 1};	q_5:	**atomic**{tq= 0; tg= 1};
	}		}

A full state description would be a quintuple, $(p_i, q_j, \text{tp}, \text{tg}, \text{tq})$. For a refresher on logical notation, see Appendix B.

(Part a) Show that each of the three variables tg, tp, tq can take only the values 0 and 1. *(2p)*

(Part b) Let I=tp+tg+tq. Prove I=1 is invariant (always true). *(3p)*

> **Notation:** Let tv stand for any one of tp, tg, tq.
> Then from **(Part a)**, $\neg(\text{tv} = 1)$ iff $\text{tv} = 0$.
> From **(Part b)**, in any state, exactly one of tp, tg, tq is 1.
> A state description need therefore only be a triple (p_i, q_j, tv), where tv is the one out of tp, tg, tq that has value 1.

(Part c) Show that $p_5 \rightarrow (\text{tp} = 1)$. *(2p).*

(Part d) Show that $p_2 \rightarrow ((\text{tp} = 0) \wedge (\text{tg} = 1))$. *(2p).*

(Part e) Show that $\neg(p_5 \wedge q_5)$ is an invariant, i.e., that the program guarantees mutual exclusion. *(3p)*

(Part f) Is there a loop of states from (p_2, q_2, tg) that includes neither p_5 nor q_5? I.e., can the system livelock, with p repeating p_2 and p_3, while q repeats q_2 and q_3? *(3p)*

(Part g) Thread q can starve thread p if a certain command by p never executes at a certain time. What command at what time? *(3p)*

Q3 (10p). The program from Q2 is repeated below for convenience. Read through Q2, including the clarifications and questions, before trying Q3.

<table>
<tr><td colspan="2" align="center">int tg= 1;</td></tr>
<tr><td>p</td><td>q</td></tr>
<tr>
<td>

int tp= 0;
while(true) {
p_1 //NCS (non-critical section)
   do{
p_2:   **atomic**{tp= tg; tg= 0};
p_3:   } **while**(tp!= 1);
p_4 //CS (critical section)
p_5:   **atomic**{tp= 0; tg= 1};
}
</td>
<td>

int tq= 0;
while(true) {
q_1 //NCS (non-critical section)
   do{
q_2:   **atomic**{tq= tg; tg= 0};
q_3:   } **while**(tq!= 1);
q_4 //CS (critical section)
q_5:   **atomic**{tq= 0; tg= 1};
}
</td>
</tr>
</table>

At the start, we expected each state to be a quintuple, $(\texttt{tp}, \texttt{tq}, \texttt{tp}, \texttt{tg}, \texttt{tq})$ where $\texttt{tp}$ and $\texttt{tq}$ can take on 3 values each, and each of $\texttt{tp}, \texttt{tg}, \texttt{tq}$ take on two values each, yielding 72 possible states. But **Q2** tells us each state can be writen $(\texttt{tp}, \texttt{tq}, \texttt{tv})$, where each of $\texttt{tp}, \texttt{tq}$ and $\texttt{tv}$ can take 3 values, so we need only deal with 27 of those 72 program states.

> **Notation:** We now also abbreviate our state notation by writing $p_i q_j$ to mean $(p_i, q_j, \texttt{tg})$, $p'_i q_j$ to mean $(p_i, q_j, \texttt{tp})$, and $p_i q'_j$ to mean $(p_i, q_j, \texttt{tq})$. **Q(2)b** tells us we won't need $p'_i q'_j$.

Here is a partial state transition table for the program above. You may assume that only 11 states are reachable from the start state $p_2 q_2$.

state $= p_i q_j$ perhaps primed		new state if p moves	new state if q moves
s1	$p_2 q_2$	$p'_3 q_2 = s8$	
s2		$p'_3 q_3 = s9$	
s3	$p_2 q'_3$		$p_2 q'_5 = s4$
s4	$p_2 q'_5$		
s5			$p_3 q'_3 = s6$
s6	$p_3 q'_3$		
s7	$p_3 q'_5$		
s8	$p'_3 q_2$		
s9	$p'_3 q_3$		
s10	$p'_5 q_2$		
s11	$p'_5 q_3$		

(Part a) Fill in the blank entries in the table.   (8p)

(Part b) Pick an unreachable state out of the 27 the abbreviated notation allows, and say why it is unreachable.   (2p)

Q4 (12p). Consider the following event loop of a "compute server" in Erlang.

```
1 server_event_loop() ->
2   receive
3     {request, Fun, Args, From} ->
4       spawn(
5         fun() ->
6           Ans = apply(Fun, Args),
7           From ! {response, Ans}
8         end)
9   end,
10 server_event_loop().
```

(Part a). Give a sequence of messages sent by the server in response to the following sequence of requests sent from *a single client*.

```
1 Me = self(),
2 server ! {request, fun exp/2, [10, 5], Me},
3 server ! {request, fun lists:append/2, [[1, 2], [3]], Me},
4 server ! {request, fun(X) -> X + X end, [8], Me},
```

where

```
1 exp(_, 0) -> 1;
2 exp(X, 1) -> X;
3 exp(X, Y) -> X * exp(X, Y - 1).
```

(3p)

(Part b). Is your answer to Part a the only possible sequence that could have been observed? *(2p)*

(Part c). What is the point of spawning new (sub-)processes for each computation? *(2p)*

(Part d). Consider the following client code for sending a sequence of requests to the server and collecting responses

```
1 %Send the messages to the server.
2 [server ! {request, fun exp/2, [Num, Exp], self()} || Num <- Nums],
3
4 %Collect answers and print them.
5 Ans = [receive {response, Ans} -> Ans end || _ <- Nums],
6 io:fwrite("~p ^ ~p = ~p~n", [Nums, Exp, Ans]).
```

Can this code go wrong? Will you get the expected output? *(2p)*

(Part e). Sketch a fix for this problem (no coding necessary, just suggest a fix in words). *(3p)*

Q5 (10p). The following recursive Java method `add()` implements vector addition for two arrays `v1` and `v2`.

```java
1 void add(int[] v1, int[] v2, int start, int end) {
2   if (end - start < 1)        return;
3   else if (end - start == 1)
4           { v1[start] = v1[start] + v2[start]; }
5   else {
6     int mid = (start + end) / 2;
7     add(v1, v2, start, mid);
8     add(v1, v2, mid, end);
9   }
10 }
```

The method adds the entries of `v1` and `v2` from index `start` to index $end - 1$ (excluding `v1[end]` and `v2[end]`), and writes the result back into `v1`.

The class `AddTask` below implements a fork/join parallelization of `add()`. The constructor of `AddTask` has the same arguments as the method `add()`; the `compute()` method computes the result and stores it in the array `v1`. See Appendix A.2 for a full code listing.

```java
1 class AddTask extends RecursiveAction {
2   private int[] v1, v2;
3   private int start, end;
4
5   AddTask(int[] v1, int[] v2, int start, int end) {
6   this.v1 = v1; this.v2 = v2;
7   this.start = start; this.end = end;
8   }
9
10  public void compute() {
11    if (end - start < 1) return;
12    else if (end - start == 1) {
13             v1[start] = v1[start] + v2[start];}
14    else {
15      int mid = (start + end) / 2;
16      AddTask t1 = new AddTask(v1, v2, start, mid);
17      AddTask t2 = new AddTask(v1, v2, mid, end);
18      t1.fork(); t2.fork();
19      t1.join(); t2.join();
20    }
21  }
22 }
```

(Part a). Draw the dependency graph for a run of `AddTask(v1, v2, 0, 8)` assuming both `v1` and `v2` have exactly 8 elements. Each node should represent a task instance, with leaf nodes representing the base cases (where `end-start <= 1`). Edges should represent parent-child relationships between tasks. How many nodes does the graph have? *(2p)*

(Part b). What is the approximate runtime of `SumTask(arr, 0, 8)` assuming there are at least 8 CPU cores and each task takes about one unit of time to perform its computation? How can this be inferred from the dependency graph? *(2p)*

(Part c). What is the maximum number of tasks that can be executed in parallel in this implementation (excluding parent tasks waiting for a child task to finish)? How can this be inferred from the dependency graph? *(2p)*

(Part d). From your dependency graph, deduce the total number of tasks started by `AddTask` when running an instance of `AddTask(v1, v2, 0, 8)`. How does this compare to the maximum number of tasks running in parallel that you computed in **Part(b)**? What does this difference correspond to? *(2p)*

(Part e). Suppose an instance of `AddTask(v1, v2, 0, 1000000)` is run on a machine with a 4-core processor, and that tasks are scheduled using an optimally configured `ForkJoinPool`. Will this result in an efficient parallel computation? Name one source of inefficiency. *(2p)*

Q6 (14p). The code on page 9 is a modified sequential singly-linked list implementation. The list stores a sorted set of integers, and the method `addIncrease` adds elements in increasing order. We tried to convert it to a lock-free thread-safe implementation using Compare and Swap.

Besides the code shown on page 9, the interface of AtomicNode contains a getter and setter for val, and a getter for next.

In the `List` class on page 9, notice that the list does not use the value field of the head node (lines 4, 11, and 18). This is intentional. The first element in the list is held in the next field of the head. The method `addIncrease` works as follows. The list is sorted in increasing order. So, if the next element in the list is larger than the value to add, then the value is added after the current element (lines 11–18). If the next element in the list is the same as the value to add, then the search stops (and the value is not added again; lines 18–20). Otherwise, the method continues to scan the list (lines 21–23, and back to line 10). Finally, if there is no next element, the value is added at the end (lines 24–26 just after the while-loop (lines 10–23) terminates).

But there are bugs in this implementation with multiple threads: the list might not be sorted, or the same value may appear multiple times.

(Part a). Explain why the implementation has this issue. *(4p)*

(Part b). Update the code of `addIncrease` to fix the implementation. You should use a CAS (compare-and-set) synchronization primitive but no semaphores or locks. Use either Java or pseudo code. *(6p)*

You benchmark the list implementation versus the usage of the original sequential list protected by a lock. You create threads that use the new (thread-safe) list and run the following code:

```
1 for (int j = 0 ; j<100 ; ++j) {
2   l.addIncrease(j);
3 }
```

You create threads that use the old sequential implementation and one additional lock and run the following code:

```
1 for (int j=0 ; j<100 ; ++j) {
2   lock.lock();
3   l.addIncrease(j-1,j);
4   lock.unlock();
5 }
```

(Part c). What happens when you run the thread-safe version vs the sequential version with the lock in the following scenarios: (i) there are 32 CPU cores and 32 threads, and (ii) there is 1 CPU core and 32 threads. *(4p)*

```
1 public class AtomicNode {
2   private int val;
3   private AtomicNode next;
4   public boolean compareAndSetNext(AtomicNode existing,
5                                    AtomicNode updated) {
6     ...
7   }
8 }
```

```
1 class List {
2   AtomicNode head;
3   public List() { // value of head not used
4     head = new AtomicNode(0,null);
5   }
6
7   public void addIncrease(int val) {
8     AtomicNode current = this.head;
9     AtomicNode next = current.getNext();
10    while (next != null) {
11      if (next.getVal() > val) {
12        AtomicNode temp;
13        do {next = current.getNext();
14            temp = new AtomicNode(val,next);
15        } while (!current.compareAndSetNext(next,temp));
16        return;
17      }
18      if (next.getVal() == val) {
19        return;
20      }
21      current = next;
22      next = current.getNext();
23    }
24    do {next = current.getNext();
25        temp = new AtomicNode(val,next);
26    } while (!current.compareAndSetNext(next,temp));
27  }
28 }
```

Figure 1: The interface AtomicNode snd the class List.

9

A Full code listings

A.1 Code for Q4

```
1  -module(q4_appendix).
2  -export([main/0]).
3
4  server_event_loop() ->                    % Server loop
5    receive
6      {request, Fun, Args, From} ->
7        spawn(
8          fun() ->
9            Ans = apply(Fun, Args),
10           From ! {response, Ans}
11         end)
12   end,
13   server_event_loop().
14
15 exp(_, 0) -> 1;
16 exp(X, 1) -> X;
17 exp(X, Y) -> X * exp(X, Y - 1).
18
19 main() ->                                 % Start and register the server
20   Pid = spawn(fun server_event_loop/0),
21   register(server, Pid),
22
23   Me = self(),                            % For Part a:
24   server ! {request, fun exp/2, [10, 5], Me},
25   server ! {request, fun lists:append/2, [[1, 2], [3]], Me},
26   server ! {request, fun(X) -> X + X end, [8], Me},
27
28   % Receive and print the responses.
29   [ receive R -> io:fwrite("~p~n", [R]) end || _ <- [1, 2, 3]],
30
31   Nums = [7,0,5,2],                       % For Part d:
32   Exp = 3,
33
34   % Send the messages to the server.
35   [server ! {request, fun exp/2, [Num, Exp], self()} || Num <- Nums],
36
37   % Collect answers and print them.
38   Ans = [receive {response, Ans} -> Ans end || _ <- Nums],
39   io:fwrite("~p ^ ~p = ~p~n", [Nums, Exp, Ans]).
```

A.2 Code for Q5

```java
1  import java.util.concurrent.*;
2
3  class AddTask extends RecursiveAction {
4    private int[] v1, v2;
5    private int start, end;
6
7    AddTask(int[] v1, int[] v2, int start, int end) {
8      this.v1 = v1; this.v2 = v2;
9      this.start = start; this.end = end;
10   }
11
12   @Override public void compute() {
13     if (end - start < 1) return;
14     else if (end - start == 1) {
15             v1[start] = v1[start] + v2[start];}
16     else {
17       int mid = (start + end) / 2;
18       AddTask t1 = new AddTask(v1, v2, start, mid);
19       AddTask t2 = new AddTask(v1, v2, mid, end);
20       t1.fork(); t2.fork();
21       t1.join(); t2.join();
22     }
23   }
24 }
25
26   public static void main(String[] args) {
27     final int N = 20;
28     int[] numbers = new int[N];
29     for (int i = 0; i < N; ++i) { numbers[i] = i; }
30
31     AddTask m = new AddTask(numbers, numbers, 0, N);
32     ForkJoinPool.commonPool().invoke(m);
33
34     for (int i = 0; i < N; ++i) {
35       System.out.println("numbers[" + i + "] = " + numbers[i]);
36     }
37   }
38 }
```

B Linear Temporal Logic (LTL) notation

1. An atomic proposition such as q_2 (process q is at label q_2) *holds for* a state s if and only if process q is at q_2 in s.

2. Let ϕ and ψ be formulas of LTL. Formulas are either atomic propositions, or are built up from other formulas using the following operators: $\neg$ for "not", $\vee$ for "or", $\wedge$ for "and", $\rightarrow$ for "implies", $\square$ for "always", and $\Diamond$ for "eventually". A convenient abbreviation is ϕ iff ψ (i.e., ϕ if and only if ψ) for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$.

   These operators have the obvious meanings, but two differ from what might be your interpretation of the names. First, $\phi \vee \psi$ ("ϕ or ψ") is false iff both ϕ and ψ are false. This is an "inclusive or", so $\phi \vee \psi$ is also true if both ϕ and ψ are true. Second, $\phi \rightarrow \psi$ ("ϕ implies ψ") is false iff ϕ is true and ψ is false. So, in particular, $\phi \rightarrow \psi$ is true if ϕ is false. The meanings of the operators $\square$ and $\Diamond$ are defined below.

3. A *path* is a possible future of the system, a possibly infinite sequence of states, each reachable from the previous state in the path. A state s *satisfies* formula ϕ if every path from s satisfies ϕ.

   A path π satisfies $\square\phi$ if ϕ holds for the first state of π, and for all subsequent states in π. The path π satisfies $\Diamond\phi$ if ϕ holds for some state in π.

   Note that $\square$ and $\Diamond$ are duals:

$$\square\phi \equiv \neg\Diamond\neg\phi \qquad \text{and} \qquad \Diamond\phi \equiv \neg\square\neg\phi.$$